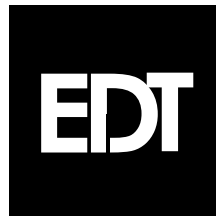


SCD-FOI

SBus Configurable DMA Fiber Optic Interface

USER'S GUIDE

008-00903-02



The information in this document is subject to change without notice and does not represent a commitment on the part of Engineering Design Team, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, without the express written agreement of Engineering Design Team, Inc.

© Copyright Engineering Design Team, Inc. 1996–1997. All rights reserved.

Refer questions or problems with this manual or the hardware or software described herein to:

Engineering Design Team, Inc.
1100 NW Compton Drive, Suite 306
Beaverton, Oregon 97006

Phone (503) 690-1234
Fax: (503) 690-1243
E-mail: info@edt.com
Web: www.edt.com

Sun, SunOS, SBus, SPARC, and SPARCstation are trademarks of Sun Microsystems, Incorporated.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

Xilinx is a registered trademark of Xilinx, Inc.

HOTLink is a trademark of Cypress Corp.

EDT and Engineering Design Team are trademarks of Engineering Design Team, Inc.

Contents

Overview	1
Installation.....	2
Installing the Hardware	2
Installing the Software.....	3
Using SunOS Version 4.1	3
Using System V Release 4 (Solaris 2.x)	4
Building the Sample Programs	4
Input and Output	5
Elements of SCD-FOI Applications	5
DMA Library Routines.....	6
Error Conditions	15
Fiber Optic Protocol.....	16
Interface Hardware	16
Writing to the Remote Device	16
Reading From the Remote Device	17
Interrupts.....	17
Registers	18
SBus Addresses	19
Output FIFO Almost Empty Threshold Register	20
Output FIFO Almost Full Threshold Register.....	20
Input FIFO Almost Empty Threshold Register	20
Input FIFO Almost Full Threshold Register	21
Current DMA Address Register	21
Next DMA Address Register.....	21
Current Count Register.....	21
Control and Next Count Register	22
Xilinx Programming Register.....	23
Command Register	23
Status Register	24
Configuration Register	24
Data_Path_Status Register	25
Board_ID Register.....	25
Local_Receiver_Error Register	25
Interrupt_Source Register.....	26
Interrupt_Enable Register.....	27
Interrupt_Status Register	27
Read_Return Register.....	27
Burst_Debug Register	28
Remote_Flag Register	28
SCD-FOI Registers as Seen by the Remote Device	28
FOI_Read_Return Register.....	28
FOI_Read_Trigger Register.....	29
FOI_Flow_Control Register	29
FOI_Burst Register	29
FOI_Flag Register.....	29

Specifications30

- SBus Compliance30
- Device Data Transfer.....30
- Software.....30
- Power30
- Environmental30
- Physical.....30

Appendix A ioctl() Parameters31

Tables

General DMA Library Routines	7
Error Codes and Conditions	15
The Output FIFO Almost Empty Threshold	20
The Output FIFO Almost Full Threshold	20
The Input FIFO Almost Empty Threshold.....	20
The Input FIFO Almost Full Threshold	21
The Current DMA Address Register	21
The Next DMA Address Register	21
The Current Count Register	21
The Control and Next Count Register.....	22
Xilinx Programming Register	23
The Command Register	23
The Status Register	24
The Configuration Register.....	24
Board_ID Register	25
The Data_Path_Status Register.....	25
The Local_Receiver_Error Register	25
The Interrupt_Source Register	26
The Interrupt_Enable Register	27
The Interrupt_Status Register	27
The Remote_Flag Register.....	28
The Burst_Debug Register.....	28
The FOI_Read_Return Register.....	28
The FOI_Read_Trigger Register	29
The FOI_Flow_Control Register	29
The FOI_Burst Register	29
The FOI_Flag Register.....	29

Overview

The SBus Configurable DMA Fiber Optic Interface (SCD-FOI) is a single-slot, bidirectional interface for SBus-based computer systems. It is designed for continuous input or output between a remote module and SBus host memory over a fiber optic link. The remote module can support such devices as scanners, plotters, imaging devices, or research prototypes. One such remote module is EDT's SDV Remote Camera Interface, which provides a configurable 16-bit parallel input-output interface to your equipment.

The SCD-FOI uses a simple EDT protocol to transfer data over the fiber optic link at speeds of up to 24 MB/second and distances of up to two kilometers. It uses 1024-byte input and output FIFOs.

The input and output FIFO buffers smooth data transfer between the SBus and the user device, as well as accommodate data during the transition from one DMA to the next. DMA transfers are queued in hardware, minimizing the amount of FIFO required.

The SCD-FOI boards fully support the requirements of the SunOS operating system.

This document explains how to install the SCD-FOI interface and driver and how to write applications for it. It is divided into the following sections:

Installation	describes how to install the board and its related software.
Input and Output	describes the software library routines available for application programs.
Fiber Optic Protocol	describes the protocol used over the fiber optic cable.
Registers	describes the hardware registers.
Specifications	lists the product specifications.
<i>ioctl()</i> Parameters	lists the <i>ioctl</i> parameters that can be useful for application programs.

Installation

Installing the SBus Configurable DMA Fiber Optic Interface is a two-step process. First you must physically install the board inside the host computer. Then you must install the software driver so that applications can access the SBus Configurable DMA Fiber Optic Interface. Hardware installation is described in the following section. Software installation is described in the section after.

Installing the Hardware

The SBus Configurable DMA Fiber Optic Interface is a single-slot SBus board. To install it, refer to your SBus host computer documentation for complete information on installing an SBus board. For example, for the following SOARCstation models information can be found in these locations:

SPARCstation Model	Location in <i>SPARCstation Installation Guide</i>
1	Appendix A: Installing Boards, Cards, and Modules
1+	Appendix A: Installing SBus Boards and SIMMs
IPX	Appendix C: Installing SBus Cards
10	Appendix B: Opening and Closing the System Unit and <i>Installing SBus Cards in Desktop SPARCstations</i> Sun part no. 800-6635-11.

Use the following procedure to install the SBus Configurable DMA Fiber Optic Interface:

CAUTION

Both the SCD-FOI and your SBus host computer contain static-sensitive components. Install the SCD-FOI at a static-free work area. If a static-free work area is not available, take the following precautions to reduce the risk of component damage:

1. Remove from the immediate area all materials that can generate or hold a static charge.
 2. Discharge yourself by touching both hands to a metal portion of the host computer's chassis before you open the host computer or open the SCD-FOI static-shielded bag.
-
-

1. Unpack the SBus Configurable DMA Fiber Optic Interface from the shipping packaging. Do not remove the SBus Configurable DMA Fiber Optic Interface from the static shielding bag until you remove all other packaging materials from the area and establish a static-free work area.
2. Install the SBus Configurable DMA Fiber Optic Interface in the SBus host, following the directions provided with the SBus host. The board can be installed in any DMA slot.

To remove the SBus Configurable DMA Fiber Optic Interface, reverse the installation procedure.

The SBus Configurable DMA Fiber Optic Interface connects to a remote device with a fiber optic cable. This cable plugs into an HP transceiver, part number HFBR 5302, which uses a standard SC Duplex connector passing 1300 nm light.

If you wish, you can use a 50-Ω coaxial cable with standard BNC connectors instead. The board is shipped with the jumpers near the transceiver strapped into the position for an optical cable—pin 1 connected to pin 2. These are the pins closest to the edge of the board. To use a coaxial cable, lift the three black plastic cases and place them over pins 2 and 3, one position further from the edge of the board.

Installing the Software

The SBus Configurable DMA Fiber Optic Interface can run on a Sun workstation using either SunOS Version 4.1.3 or 4.1.4, or Solaris 2.x (System V Release 4, or SVR 4). The installation procedures differ. Both are given below.

Using SunOS Version 4.1

If you are using SunOS Version 4.1.3 or 4.1.4, use the following procedure to install the SBus Configurable DMA Fiber Optic Interface driver:

1. Become root or superuser.
2. Change to the directory in which you wish to install the SBus Configurable DMA Fiber Optic Interface driver.
3. Place the diskette that came with the SBus Configurable DMA Fiber Optic Interface into the diskette drive.
4. The SBus Configurable DMA Fiber Optic Interface driver and related files are included on a diskette in *tar* format. To copy them to your hard disk, enter:

```
tar xvf /dev/rfd0
```

5. The *tar* program extracts a number of files. (The list of files distributed is provided in the section entitled **Included Files**.) The SBus Configurable DMA Fiber Optic Interface diskette contains versions of the SBus Configurable DMA Fiber Optic Interface driver for a variety of Sun platforms and versions of the Sun operating system. The installation program installs the correct driver based on the host platform and operating system version.
6. To install the driver, enter:

```
make install
```

The makefile provided installs and loads the SBus Configurable DMA Fiber Optic Interface driver.

7. During the installation, the following question appears on the display:

```
Automatically load the SCD-FOI driver during each reboot? [y|n] (y):
```

Entering *y* (or simply typing <Return>) causes the SBus Configurable DMA Fiber Optic Interface driver to be loaded whenever you reboot your host computer. If you respond with *n*, you must manually reload the driver after rebooting. To do so, enter:

```
make load
```

8. During the installation, the following question appears on the display:

```
How many SCD-FOI devices do you want? (1):
```

You can install as many SBus Configurable DMA Fiber Optic Interfaces in your system as you have DMA SBus slots available. Enter the number corresponding to the number of SBus Configurable DMA Fiber Optic Interfaces you have installed in your system. If you simply type <Return>, one SCD-FOI device entry is installed.

NOTE: If you anticipate installing more than one SBus Configurable DMA board into your system, install as many SBus Configurable DMA board device entries as you will ultimately require. The extra device entries will do no harm and will be there when you need them, saving you a step.

9. If the SBus Configurable DMA Fiber Optic Interface has not been installed inside the host computer, or has been installed incorrectly, the following message appears on the display:

```
Can't load this module
```

If you see this message, go back to the section entitled **Installing the Hardware** and reinstall the board.

To unload the SBus Configurable DMA Fiber Optic Interface driver:

1. Change to the directory in which you placed the SBus Configurable DMA Fiber Optic Interface files, if you are not already there.
2. Become root or superuser.
3. Enter:

```
make unload
```

Using System V Release 4 (Solaris 2.x)

If you are using Sun System V Release 4 (Solaris 2.x), use the following procedure to install the SBus Configurable DMA Fiber Optic Interface driver:

1. Become root or superuser.
2. Place the diskette that came with the SBus Configurable DMA Fiber Optic Interface into the diskette drive.
3. Enter:

```
volcheck  
pkgadd -d /vol/dev/aliases/floppy0 EDTscdfoi
```

To remove the SBus Configurable DMA Fiber Optic Interface driver:

1. Become root or superuser.
2. Enter:

```
pkgrm EDTscdfoi
```

For further details, consult your Solaris 2.0 documentation, or call Engineering Design Team, Inc.

Building the Sample Programs

To build any of the example programs, enter the command:

```
make file
```

where *file* is the name of the example program you wish to install.

To build and install all the example programs, simply enter the command:

```
make
```

All example programs display a message that explains their usage when you enter their names without parameters.

For a complete, up-to-date listing of the files included on the SCD-FOI driver release diskette, see the *readme* file.

Input and Output

The SCD-FOI device driver can perform two kinds of DMA transfers: continuous and noncontinuous. For noncontinuous transfers, the driver uses DMA system calls *read()* and *write()*. Each *read()* and *write()* system call allocates kernel resources, during which time DMA transfers are interrupted.

To perform continuous transfers, use the ring buffers. The ring buffers are a set of buffers that applications can access continuously, reading and writing as required. When the last buffer in the set has been accessed, the application then cycles back to the first buffer. See `edt_configure_ring_buffers()` for a complete description of the ring buffer parameters that you can configure. See the sample programs `simple_getdata.c` and `simple_putdata.c` distributed with the driver for examples of using the ring buffers.

Elements of SCD-FOI Applications

Applications for performing continuous transfers typically include the following elements:

```
#include "libedt.h"

main()
{
    EdtDev *edt_p = edt_open("scd", 0) ;
    char *buf_ptr; int outfd = open("outfile", 1) ;

    /* Configure a ring buffer with four 1MB buffers */
    edt_configure_ring_buffers(edt_p, 1024*1024, 4, EDT_READ, NULL) ;
    edt_start_buffers(edt_p, 0) ; /* 0 starts unlimited buffer DMA */
    /* This loop will capture data indefinitely, but the write()
     * (or whatever processing on the data) must be able to keep up. */
    while ((buf_ptr = edt_wait_for_buffer(edt_p, 1)) != NULL)
        write(outfd, buf_ptr, 1024*1024) ;

    edt_close(edt_p) ;
}
```

Applications for performing noncontinuous transfers typically include the following elements:

```
#include "libedt.h"

main()
{
    EdtDev *edt_p = edt_open("scd", 1) ;
    char buf[1024] ;
    int numbytes, outfd = open("outfile", 1) ;
    /*
     * Because read()s are noncontinuous, unless is there hardware
     * handshaking there will be gaps in the data between each read().
     */
    while ((numbytes = edt_read(edt_p, buf, 1024)) > 0)
        write(outfd, buf, numbytes) ;

    edt_close(edt_p) ;
}
```

You can use ring buffer mode for real-time data capture using a small number of (typically 1 MB) buffers configured in a round-robin data FIFO. During capture, the application must be able to transfer or process the data before data acquisition wraps around and overwrites the buffer currently being processed.

The example below shows real-time data capture using ring buffers, although it includes no error-checking. In this example, `process_data(bufptr)` must execute in the same amount of time it takes DMA to fill a single buffer, or faster.

```
#include "libedt.h"

main()
{
    EdtDev *edt_p = edt_open("scd", 0) ;

    /* Configure four 1 MB buffers:
     *     one for DMA
     *     one for the second DMA register on most EDT boards
     *     one for "process_data(bufptr)" to work on
     *     one to keep DMA away from "process_data()"
     */
    edt_configure_ring_buffers(edt_p, 0x100000, 4, EDT_READ, NULL) ;
    edt_start_buffers(edt_p, 0) ; /* 0 starts unlimited buffer DMA */
    for (;;)
    {
        char *bufptr ;

        /* Wait for each buffer to complete, then process it.
         * The driver continues DMA concurrently with processing.
         */
        bufptr = edt_wait_for_buffers(edt_p, 1) ;
        process_data(bufptr) ;
    }
}
```

Use the "`-D_REENTRANT -ledt -lthread`" options to compile and link the library file *libedt.a* with your program. See the makefile and example programs provided for examples of compiling programs using the library routines.

DMA Library Routines

The DMA library provides a set of consistent routines across many of the EDT products, with simple yet powerful ring-buffered DMA capabilities. **Table 1, DMA Library Routines** lists the general DMA library routines. In addition, if driver-specific library routines exist, they can be found in a table thereafter.

The sections that follow describe the DMA library routines in an order corresponding roughly to their general usefulness.

Routine	Description
edt_open	Opens the SCD-FOI for application access.
edt_close	Terminates access to the SCD-FOI and releases resources.
edt_read	Single, application-level buffer read from the SCD-FOI.
edt_write	Single, application-level buffer write to the SCD-FOI.
edt_configure_ring_buffers	Configures the ring buffers.
edt_disable_ring_buffers	Stops DMA transfer, disables ring buffers and releases resources.
edt_start_buffers	Begins DMA transfer from or to specified number of buffers.
edt_stop_buffers	Stops DMA transfer after the current buffer(s) complete(s).
edt_wait_for_buffers	Blocks until the specified number of buffers have completed.
edt_next_writebuf	Returns a pointer to the next buffer scheduled for output DMA.
edt_check_for_buffers	Checks whether the specified number of buffers have completed without blocking.
edt_wait_for_next_buffer	Waits for the next buffer that completes DMA.
edt_ring_buffer_overflow	Detects ring buffer overrun which may have corrupted data.
edt_buffer_addresses	Returns an array of addresses referencing the ring buffers.
edt_abort_dma	Cancels the current DMA, resets pointers to the current buffer
edt_abort_current_dma	Cancels the current DMA, moves pointers to the next buffer.
edt_done_count	Returns absolute (cumulative) number of completed buffers.
edt_reset_ring_buffers	Stops DMA in progress and resets the ring buffers.
edt_microsleep	Process sleeps for the specified number of microseconds.

Table 1. General DMA Library Routines

edt_open

Description

Opens the specified SCD-FOI and sets up the device handle.

Syntax

```
EdtDev *edt_open(char *devname, int unit) ;
```

Arguments

devname a string with the name of the EDT board.

unit specifies the device unit number

Return

A handle of type (`EdtDev *`), or `NULL` if error. (The structure (`EdtDev *`) is defined in `libedt.h`.) If an error occurs, check the `errno` global variable for the error number. The device name for the SCD-FOI is "scd". Once opened, the device handle may be used to perform I/O using `edt_read()`, `edt_write()`, `edt_configure_ring_buffers()`, and other input-output library calls.

edt_close

Description

Shuts down all pending I/O operations, closes the device and frees all driver resources.

Syntax

```
int edt_close(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_read

Description

Performs a read on the SCD-FOI. The UNIX 2 GB file offset bug is avoided during large amounts of input or output, that is, reading past 2^{31} bytes does not fail. This call is not multibuffering, and no transfer is active when it completes.

Syntax

```
int edt_read(EdtDev *edt_p, void *buf, int size);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*

buf address of buffer to read into

size size of read in bytes

Return

The return value from *read*, normally the number of bytes read; -1 is returned and *errno* is set by *read* on error.

edt_write

Description

Perform a write on the SCD-FOI. The UNIX 2 GB file offset bug is avoided during large amounts of input or output; that is, writing past 2^{31} does not fail. This call is not multibuffering, and no transfer is active when it completes.

Syntax

```
int edt_write(EdtDev *edt_p, void *buf, int size);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*

buf address of buffer to write from

size size of write in bytes

Return

The return value from *write*; -1 is returned and *errno* is set by *write* on error.

edt_configure_ring_buffers

Description

Configures the SBus Configurable DMA Fiber Optic Interface ring buffers. Any previous configuration is replaced, and previously allocated buffers are released. Buffers can be allocated and maintained within the SBus Configurable DMA Fiber Optic Interface library or within the user application itself.

Syntax

```
int edt_configure_ring_buffers(EdtDev *edt_p, int bufsize, int nbufs,
                               int data_output, void *bufarray[]);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*

bufsize size of each buffer. For optimal efficiency, allocate a value approximating throughput divided by 20: that is, if transfer occurs at 20 MB per second, allocate 1 MB per buffer. Buffers significantly larger or smaller can overuse memory or lock the system up in processing interrupts at this speed.

nbufs number of buffers. Must be 1 or greater. Four is recommended for most applications.

data_direction Indicates whether this connection is to be used for input or output. Only one direction is possible per device or subdevice at any given time:
EDT_READ = 0
EDT_WRITE = 1

bufarray If NULL, the library will allocate a set of page-aligned ring buffers. If not NULL, this argument is an array of pointers to application-allocated buffers; these buffers must match the size and number of buffers specified in this call and will be used as the ring buffers.

Return

0 on success; -1 on error. If all buffers cannot be allocated, none are allocated and an error is returned. The global variable *errno* is set on error.

edt_disable_ring_buffers

Description

Disables the SBus Configurable DMA Fiber Optic Interface ring buffers. Pending DMA is cancelled and all buffers are released.

Syntax

```
int edt_disable_ring_buffers(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_start_buffers

Description

Starts DMA to the specified number of buffers.

Syntax

```
int edt_start_buffers(EdtDev *edt_p, int bufnum);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*

bufnum Number of buffers to release to the driver for transfer. An argument of 0 puts the driver in free running mode, and transfers run continuously until **edt_stop_buffers()** is called.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_stop_buffers

Description

Stops DMA transfer after the current buffer has completed. Ring buffer mode remains active, and transfers will be continued by calling **edt_start_buffers()**.

Syntax

```
int edt_stop_buffers(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_wait_for_buffers

Description

Blocks until the specified number of buffers have completed.

Syntax

```
void *edt_wait_buffers(EdtDev *edt_p, int bufnum);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*

bufnum buffer number for which to block. Completed buffers are numbered cumulatively starting with 0 when the SCD-FOI is opened.

Return

Address of last completed buffer on success; NULL on error. If an error occurs, check the *errno* global variable for more information.

NOTE: If the ring buffer is in free-running mode and the application cannot process data as fast as it is acquired, DMA will wrap around and overwrite the referenced buffer. The application must ensure that the data in the buffer is processed or copied out in time to prevent overrun.

For an example of real-time data capture using ring buffers, see the example on page 6.

edt_next_writebuf

Description

Returns a pointer to the next buffer scheduled for output DMA, in order to fill the buffer with data.

Syntax

```
void *edt_next_writebuf(EdtDev *edt_p) ;
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

Returns a pointer to the buffer, or NULL on failure. Sets *errno* on failure.

edt_check_for_buffers

Description

Checks whether the specified number of buffers have completed without blocking.

Syntax

```
int edt_check_for_buffers(EdtDev *edt_p, count);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.
nbufs number of buffers.
count number of buffers. Must be 1 or greater. Four is recommended.

Return

Returns the address of the ring buffer corresponding to *count* if it has completed DMA, or NULL if *count* buffers are not yet complete.

NOTE: If the ring buffer is in free-running mode and the application cannot process data as fast as it is acquired, DMA will wrap around and overwrite the referenced buffer. The application must ensure that the data in the buffer is processed or copied out in time to prevent overrun.

edt_wait_for_next_buffer

Description

Waits for the next buffer that finishes DMA. Depending on how often this routine is called, buffers that have already completed DMA might be skipped.

Syntax

```
void *edt_wait_for_next_buffer(EdtDev *edt_p) ;
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

Returns a pointer to the buffer, or NULL on failure. Sets *errno* on failure.

edt_ring_buffer_overflow

Description

Returns true (1) when DMA has wrapped around the ring buffer and overwritten the buffer which the application is about to access. Returns false (0) otherwise.

Syntax

```
int edt_ring_buffer_overflow(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

1 (true) when overflow has occurred, corrupting the current buffer, 0 false() otherwise..

edt_buffer_addresses

Description

Returns an array containing the addresses of the ring buffers.

Syntax

```
void **edt_buffer_addresses(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

An array of pointers to the ring buffers allocated by the driver or the library. The array is indexed from zero to *n*-1 where *n* is the number of ring buffers set in **edt_configure_ring_buffers()**.

edt_abort_dma

Description

Stops any transfers currently in progress, resets the ring buffer pointers to restart on the current buffer.

Syntax

```
int edt_abort_dma(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_abort_current_dma

Description

Stops the current transfers, resets the ring buffer pointers to the next buffer.

Syntax

```
int edt_abort_dma(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_done_count

Description

Returns the cumulative count of completed buffer transfers in ring buffer mode.

Syntax

```
int edt_done_count(EdtDev *edt_p);
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

Return

The number of completed buffer transfers. Completed buffers are numbered consecutively starting with 0 when **edt_configure_ring_buffers()** is invoked. The index of the ring buffer most recently completed by the driver equals the number returned modulo the number of ring buffers. -1 is returned if ring buffer mode is not configured and the global variable *errno* is set.

edt_reset_ring_buffers

Description

Stops any DMA currently in progress, then resets the ring buffer to start the next DMA at *bufnum*.

Syntax

```
int edt_reset_ring_buffers(EdtDev *edt_p, int bufnum) ;
```

Arguments

edt_p SCD-FOI device handle returned from *edt_open*.

bufnum The index of the ring buffer at which to start the next DMA.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

edt_microsleep**Description**

Causes the process to sleep for the specified number of microseconds.

Syntax

```
int edt_microsleep(u_int usecs) ;
```

Arguments

usecs The number of microseconds for the process to sleep.

Return

0 on success; -1 on error. If an error occurs, check the *errno* global variable for the error number.

Error Conditions

The table below shows error codes for the SCD-FOI and the error condition represented by each code.

Error Code	Failing Command	Error Condition
EINVAL	<i>ioctl()</i>	An invalid command was used, or an invalid length was specified for the buffer
EFAULT		An argument points outside the allocated address space.

Table 2. Error Codes and Conditions

Fiber Optic Protocol

This section is intended for the hardware designer of a remote interface, or anyone else who would like a sense of the capabilities of the SBus Configurable DMA Fiber Optic Interface. It describes the interface hardware, the protocol for reading from and writing to the remote device, and the implementation of interrupts and error detection.

Interface Hardware

The Cypress HOTLink™ CY7B923 transmitter and CY7B933 receiver chips, and the HP HFBR 5302 transceiver (PECL to or from 1300 nm light) are used to implement a byte-serial interface that can transmit over optical fiber or coaxial cable. Transmission over the fiber is in one direction only; a second transmitter, receiver, and optical fiber are provided for the return data path. Each runs the same protocol.

Transmitted characters consist of eight data bits plus one Special Character/Data Select bit, which indicates if the other eight are a control code or a data byte. The protocol described in this section uses two control codes. A third is the pad byte (control code of 0x05 plus the Special Character/Data Select bit asserted), which the Cypress hardware sends automatically when it is idle in order to synchronize the remote receiver. Pad bytes are further described in the *Cypress HOTLink User's Guide*. Online information is available at <http://www.cypress.com/cypress/prodgate/datacom/cy7b923.html>.

Writing to the Remote Device

To write to the remote node, use the FOIADR and FOIDAT control codes. These always work as a pair. The FOIADR opcode is followed by one data byte that specifies an address on the remote node to which the data is to be written. It must be followed by a FOIDAT opcode and then the data byte to write to the specified address.

FOIADR data	c01	(Control code of 0x01 plus the Special Character/Data Select bit asserted) (8-bit address)
FOIDAT data	c00	(Control code of 0x00 plus the Special Character/Data Select bit asserted) (8 bits of data)

For example, to write a value of 0x34 to the register at address 0x12 on the remote device, send the following over the channel:

```
c01 0x12 c00 0x34
```

The protocol allows idle periods to be inserted between these codes.

In the case of burst writes, an arbitrary number of data bytes is allowed. The burst write register is the SCD-FOI register at address 0x0D. Once the channel is set up, bytes written to this register from the remote device can be transferred into the host computer's main memory using SBus DMA (direct memory access). This capability is symmetrical; bytes written to the burst-write register on the remote device (perhaps by means of SBus DMA hardware on the SCD-FOI board) are made available as a data stream to whatever equipment you may have connected to the remote device. For example:

```
c01 0x0D c00 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ...
```

If you need flow control, the application receiving the data can write to a register on the sending end, telling it when to start or stop the burst.

NOTE: The receiving end must be able to accept all the data bytes arriving during the time it takes for the stop command to propagate to the sender and back over the fiber optic cable, plus any internal response time within the sender.

Reading From the Remote Device

Because each fiber optic channel operates in one direction only, read data is returned by a write operation over the return channel, using the same protocol.

To initiate a read, write to the Read_Trigger register at address 0x01 on the remote device. The value you write to this register is the address of the register to be read. The data is returned when the remote device writes back to local register 0x00 (the Read_Data register) with the requested data.

For example, to read the register at address 0x54 of the remote device, send the following from the local device:

```
c01 0x01 c00 0x54
```

Assuming that the data word in the register at 0x54 on the remote device is 0x32, the remote device then returns the following to write to the local Read_Data register:

```
c01 0x00 c00 0x32
```

You can initiate read operations in other ways. For example, a design can adopt the convention of strobing a control register bit to cause a status register to be written back to register 0. Or setting the GO bit of a register on a remote camera interface could cause a large block of data to be streamed back to the local burst register address.

NOTE: To avoid overwriting previously written data, do not write to the Read_Data register at address 0x00 of the remote device unless specifically requested by the remote device.

Interrupts

Interrupts are initiated from the remote device through writes to a designated register on the local device. The interrupt is serviced when the local device writes to a remote device register.

Registers

The SBus Configurable DMA Fiber Optic Interface is configured and controlled through SBus registers and an FCode PROM. Applications can access the SCD-FOI registers through the DMA library routines, or if necessary by means of *ioctl()* calls with SCD-FOI-specific parameters, as defined in the file *scd.h*.

NOTE: All registers except the Xilinx interface control registers (at addresses 0x0001.0080 and above) are initialized and manipulated by the SCD-FOI driver. User applications do not ordinarily need to write these registers.

Registers between 0x0000.0000 and 0x0001.0050 are permanently implemented by the hardware. They are used to set up the DMA channel to the SBus and to program the Xilinx field-programmable gate array (FPGA). The registers between 0x0001.0080 and 0x0001.009F are implemented in the Xilinx FPGA and are available only after the configuration file has been downloaded to it through the Xilinx programming register at 0x0001.0050 using the software utility *sfoiload*. You must do so each time the SCD-FOI is powered on.

The registers between 0x0001.00C0 and 0x0001.00FF are mapped to physical registers implemented in the remote device at the other end of the fiber optic cable, using the local Xilinx FPGA. Therefore, you must configure the Xilinx before you can communicate with the remote device.

You can read and write registers implemented on the remote device by accessing the corresponding SBus addresses; for complete information, see the manual for the remote device you have installed.

You can read registers on the remote device, but such operations can take tens of microseconds due to a long cable or competing traffic, and the SBus *read* can time out before the operation completes. The SBus specification allots 255 SBus clock cycles before a *read* time-out. The period of time this represents changes according to the speed of the SBus clock; see the Sun documentation for your host computer for specific time-out time periods.

If delays are too long to read the remote registers directly, you can read them by writing to the *Read_Trigger* register on the remote device, then reading the local *Read_Return* register after the data has arrived.

SBus Addresses

The addresses listed in the table below are offsets from the SBus slot base addresses. Obtain the SBus base address from the SBus host documentation. The figure below describes the SBus Configurable DMA board registers in detail.

0x0001.00C0– 0x0001.00FF	mapped to registers on remote device			
	not used			
0x0001.008C	Read_Return	Burst_Debug	not used	Remote_Flag
0x0001.0088	Interrupt_Source	Interrupt_Enable	Interrupt_Status	not used
0x0001.0084	not used	Board_ID	Local_Receiver_Error	not used
0x0001.0080	Command	Status	Configuration	Data_Path_Status
	not used			
0x0001.0050	Xilinx programming			
0x0001.004C	Control and next count			
0x0001.0048	Current count			
0x0001.0044	Next DMA address			
0x0001.0040	Current DMA address			
	not used			
0x0001.000C	Input FIFO almost full threshold			
0x0001.0008	Input FIFO almost empty threshold			
0x0001.0004	Output FIFO almost full threshold			
0x0001.0000	Output FIFO almost empty threshold			
				ROM byte 0x7FFF
	ROM			
0x0000.0000	ROM byte 0			
Byte	0	1	2	3
Word	0		1	

Figure 1. SBus Addresses

Output FIFO Almost Empty Threshold Register

The output FIFO almost empty threshold register is a 32-bit register at address 0x10000. This register allows you to specify an 8-bit number representing the number of data items in the output FIFO required to set the “almost empty” flag. The SCD-FOI board includes two output FIFOs, and both must be set to the same threshold with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the output FIFO required to set the “almost empty” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the output FIFO required to set the “almost empty” flag.

Table 3. The Output FIFO Almost Empty Threshold

Output FIFO Almost Full Threshold Register

The output FIFO almost full threshold register is a 32-bit register at address 0x10004. This register allows you to specify an 8-bit number representing the number of data items in the output FIFO required to set the “almost full” flag. The SCD-FOI board includes two output FIFOs, and both must be set to the same threshold with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the output FIFO required to set the “almost full” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the output FIFO required to set the “almost full” flag.

Table 4. The Output FIFO Almost Full Threshold

Input FIFO Almost Empty Threshold Register

The input FIFO almost empty threshold register is a 32-bit register at address 0x10008. This register allows you to specify an 8-bit number representing the number of data items in the input FIFO required to set the “almost empty” flag. The SCD-FOI board includes two input FIFOs, and both must be set to the same threshold with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the input FIFO required to set the “almost empty” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the input FIFO required to set the “almost empty” flag.

Table 5. The Input FIFO Almost Empty Threshold

Input FIFO Almost Full Threshold Register

The input FIFO almost full threshold register is a 32-bit register at address 0x1000C. This register allows you to specify an 8-bit number representing the number of data items in the input FIFO required to set the “almost full” flag. The SCD-FOI board includes two input FIFOs, and both must be set to the same threshold with the same write operation. Therefore, bits 0–7 and bits 16–23 must hold the same value. Bits 8–15 and bits 24–31 are not used; set them to zero.

Bit	Description
OE31–24	Set to 0.
OE23–16	The number of data items in the input FIFO required to set the “almost full” flag.
OE15–8	Set to 0.
OE7–0	The number of data items in the input FIFO required to set the “almost full” flag.

Table 6. The Input FIFO Almost Full Threshold

Current DMA Address Register

The current DMA address register is a 32-bit read-only register at address 0x10040.

Bit	Description
A31–A22	The 4 MB page addressed by the DMA.
A21–A2	When read, the next address to access on the SBus.
A1–A0	Always 0.

Table 7. The Current DMA Address Register

Next DMA Address Register

The next DMA address register is a 32-bit register at address 0x10044.

Bit	Description
NA31–22	Show or store the 4 MB page addressed by the next DMA. When the next DMA starts, this value is copied into the corresponding bits of the current DMA address register.
NA21–2	Sets the address the next DMA will use. When the next DMA starts, this value is copied into the corresponding bits of the current DMA address register.
NA1–0	Set to 0.

Table 8. The Next DMA Address Register

Current Count Register

The current count register is a 32-bit read-only register at address 0x10048.

Bit	Description
C31–22	Always 0.
C21–C2	When read, these bits display how many words remain in the DMA transfer currently in progress.
C1–0	Always 0.

Table 9. The Current Count Register

Control and Next Count Register

The control and next count register is a 32-bit register at address 0x1004C.

Bit	SCD-FOI_	Description
D31	INT	A read-only status bit. A value of 1 indicates the SCD-FOI-20 is asserting an SBus interrupt.
D30	DMA_INPROG	A read-only status bit. A value of 1 indicates a DMA in progress.
D29	DMA_START	A value of 1 enables DMA transfer.
D28	EN_INTFC	A value of 1 enables interface interrupt.
D27	EN_EODMA	A value of 1 enables end-of-DMA interrupt.
D26	CANCEL	Cancels SBus DMA in progress.
D25	DMA_DIR_MSK	DMA direction: a value of 1 reads host memory, 0 writes it.
D24	BURST_EN	A value of 1 enables burst transfer.
D23–22		The burst size for burst transfers. Values are mapped as follows:
	2WD_BURST	0 = 2-word burst transfer
	4WD_BURST	1 = 4-word burst transfer
	8WD_BURST	2 = 8-word burst transfer
	16WD_BURST	3 = 16-word burst transfer
D21–2	SIZ_MSK	How many words to transfer in the next DMA transfer. When the next DMA starts, this value is copied into the corresponding bits of the current count register.
D1–0	CNT_MSK	Always 0.

Table 10. The Control and Next Count Register

Xilinx Programming Register

The Xilinx programming register is a 32-bit register at address 0x10050. The Xilinx chip is a field-programmable integrated circuit used to implement the SCD-FOI interface or to test the board. The Xilinx programmable IC is programmed serially.

NOTE: All application-specific registers reside in the Xilinx IC. In order to access those registers, the SCD-FOI board requires that the Xilinx be loaded with a program that defines them. If the Xilinx is not loaded, or loaded with an incorrect program, those registers are inaccessible. The Xilinx IC is programmed when the SCD-FOI driver is loaded, or by the application.

Bit	X_	Description
D31–2		not used
D1	PROG (<i>write</i>) INIT (<i>read</i>)	Sets the Xilinx IC into programming mode, in which it is able to accept program data. To do so: <ol style="list-style-type: none"> 1. Set this bit to 1. 2. Clear this bit to 0. 3. Wait until reading this bit produces a value of 1, indicating that the IC is ready to accept a program.
D0	DATA_MSK (<i>write</i>) DONE (<i>read</i>)	The bit used for the serial data stream containing the program. When read, a value of 1 indicates that the IC is done accepting the program data.

Table 11. Xilinx Programming Register

Command Register

The command register is an 8-bit write-only register at address 0x0001.0080. It is implemented with strobe bits; you do not need to clear it explicitly.

Bit	SCD-FOI_	Description
D0	RSTIF	Initialize the interface.
D1	RSTFIFO	Initialize the FIFO.
D2	not used	
D3	CLRFTO	Clear FOI_TIMEOUT (bit 3 of the Interrupt_Source register).
D4–7	not used	Write with zeroes only.

Table 12. The Command Register

Status Register

The status register is an 8-bit read-only register at address 0x0001.0081.

Bit	SCD-FOI_	Description
D0–4		not used—may return invalid data
D5	RTNVALID	Data in the Remote Read register is valid.
D6	FLAGVALID	Data in the Remote Flag register is valid.
D7		not used

Table 13. The Status Register

Configuration Register

The configuration register is an 8-bit read-write register at address 0x0001.0082. Reading this register returns the last data written.

Bit	SCD-FOI_	Description
D0	DMA_IN	A value of 1 indicates that data is entering the SCD-FOI from the remote device. A value of 0 indicates that data is leaving the SCD-FOI to the remote device.
D1	DMA_ENAB	Set to 1 to enable SCD-FOI interface DMA. This bit is set after all other registers have been set up for DMA. Reset this bit to reset the direction or flags.
D2	DMA_BOTH	Turn on DMA in both directions—for internal debugging purposes only.
D3	not used	
D4	RAWOUT	Set to 1 to allow raw data and control codes to be sent out, ignoring the protocol. Write to an even remote FOI register address, such as FOI_Flow_Control, to send a data byte. Write to an odd remote FOI register address, such as FOI_Burst, to send a control code.
D5	not used	
D6	BISTEN	Enable built-in self-test—for internal debugging purposes only.
D7	SELB	Select local loopback—for internal debugging purposes only.

Table 14. The Configuration Register

Data_Path_Status Register

The Data_Path_Status register is an 8-bit read-only register at address 0x0001.0083.

Bit	SCD-FOI_	Description
D0	OF_NOT_EMP	If this bit is set, the output FIFO is not empty.
D1	IF_NOT_EMP	If this bit is set, the input FIFO is not empty.
D2	not used	
D3	OVERFLOW	True if the remote device sent data faster than the host could receive it, overflowing the input FIFO. Reset this bit with the RSTFIFO bit in the command register.
D4–7	not used	Used for internal debugging purposes only

Table 15. The Data_Path_Status Register

Board_ID Register

The Board_ID register is an 8-bit read-only register at address 0x0001.0085. Driver software on the host computer can read this register to determine the installed version of Xilinx firmware (.rft file).

Bit	Description
B10–7	Firmware version identification.

Table 16. Board_ID Register

Local_Receiver_Error Register

The Local_Receiver_Error register is an 8-bit read-only register at address 0x0001.0086 which shows error states detected by the local HOTLink receiver. (These error states are explained in Table 2 on page 15.) Bits set in this register remain set until explicitly cleared. Clear this register by strobing the CLRRCVE bit of the command register. Then read it once again to detect the result of any write operation that may have occurred before it received the CLRRCVE strobe.

Bit	SCD-FOI_	Description
D0	NOCARRIER	True if the local device detects no carrier on its incoming cable.
D1	RVSEERR	True if the local device receives a violation symbol on its incoming cable.
D2	CRCERR	True if the local device sees that its received data did not checksum correctly after receiving a FOICRC command from the remote device.
D3–7		not used

Table 17. The Local_Receiver_Error Register

Interrupt_Source Register

The Interrupt_Source register is an 8-bit read-only register at address 0x0001.0088, used to implement interrupts on behalf of the remote device. Write all ones to clear this register to all zeroes.

Bit	SCD-FOI_	Description
D0	RCVERR	True if any of the local HOTLink receiver error conditions reported in the Local Receiver Error register are true. Cleared when Local Receiver Error register is read.
D1	not used	
D2	FLAGVALID	True if the remote device has written to the Remote Flag register. Cleared when the Remote Flag register is read.
D3	FOI_TIMEOUT	True if the remote device did not respond to a FOI read within 8 μ s. If so, the SBus read that initiated the FOI read will have aborted, returning invalid data to the host. Cleared by bit 3 (CLRFTO) of the Command register.
D4–7		not used

Table 18. The Interrupt_Source Register

Interrupt_Enable Register

The Interrupt_Enable register is an 8-bit read-write register at address 0x0001.0089. Reading this register returns the last data written.

Bit	SCD-FOI_	Description
D0	RCVERR	Enable interrupts on the RCVERR bit in the Interrupt Source register.
D1	not used	
D2	FLAGVALID	Enable interrupts on the RMTFLG bit in the Interrupt Source register.
D3	FOI_TIMEOUT	Enable interrupts on the FOI_TIMEOUT bit in the Interrupt Source register.
D4–7		not used

Table 19. The Interrupt_Enable Register

Interrupt_Status Register

The Interrupt_Status register is an 8-bit read-only register at address 0x0001.008A. Reading this register returns the result of ANDing the Interrupt_Enable and Interrupt_Source bits, indicating which bit is initiating an SBus interrupt.

Bit	SCD-FOI_	Description
D0	RCVERR	The result of ANDing the corresponding bits from the Interrupt_Enable and Interrupt_Source registers.
D1	not used	
D2	FLAGVALID	The result of ANDing the corresponding bits from the Interrupt_Enable and Interrupt_Source registers.
D3	FOI_TIMEOUT	The result of ANDing the corresponding bits from the Interrupt_Enable and Interrupt_Source registers.
D4–7		not used

Table 20. The Interrupt_Status Register

Read_Return Register

The Read_Return register is an 8-bit read-only register at address 0x0001.008C. This register holds data written to the FOI read register by the remote device in response to a read command. This is usually initiated from the local device by writing to the Read Trigger register of the remote device. When the FOI_Read register is loaded by the remote device, the RTNVALID bit of the status register is set to one until the Read_Return register is next accessed by the host.

Normally, driver or user software need not access either this register or the RTNVALID bit, because the Xilinx firmware automatically handles reads of registers on the remote device. However, a fiber optic cable longer than one kilometer will cause the SBus to timeout on reads. In this case, read the remote device by writing to its Read Trigger register.

Bit	Description
D0–7	Read return data from remote device.

Burst_Debug Register

The Burst_Debug register is an 8-bit write-only register at address 0x0001.008D. It is intended for internal debugging purposes only. When the host writes to this register, the data goes to the remote device as a burst write. The remote device can read the contents of this register through the FOI_Burst register documented on page 29.

Bit	Description
D0–7	Data to remote device.

Table 21. The Burst_Debug Register

Remote_Flag Register

The Remote_Flag register is an 8-bit read-only register at address 0x0001.008F. When the remote device writes to this register, the SBus host can read it. When the remote flag register is loaded by the remote device, the FLAGVALID bit of the status register is set to one until the remote flag register is next read by the host. The FLAGVALID bit is available in the Interrupt Source register described on page 26 and can be used as the source of an interrupt to the host.

Bit	SCD-FOI_	Description
D0–7	RMTF0–7	Status from the remote device.

Table 22. The Remote_Flag Register

SCD-FOI Registers as Seen by the Remote Device

The data transmission protocol used by the SCD-FOI (and described in the section entitled “Fiber Optic Protocol” beginning on page 16) is symmetrical; however, its uses typically are not. Ordinarily the remote device does not issue commands; it merely responds to commands from the SCD-FOI board in the host computer. Therefore, few SCD-FOI registers are accessible from the remote device over the fiber optic cable.

In addition to the registers described below, a FOIACK command from the remote device deposits a byte into an 8-bit register. The host can read an accumulated version of these bytes in the SBus Remote Ack Error register.

FOI_Read_Return Register

The FOI_Read_Return register is an 8-bit register at address 0x00. Data written to this register can be read from the SBus by means of the Read_Return register. The Xilinx firmware handles this register automatically, as it supports SBus reads of remote registers mapped to SBus addresses 0x0001.00C0 through 0x0001.00FF.

Bit	Description
D0–7	Return read data from remote device.

Table 23. The FOI_Read_Return Register

FOI_Read_Trigger Register

The FOI_Read_Trigger register is an 8-bit register at address 0x01. When the remote device writes to this register with the address of an SCD-FOI register, the SCD-FOI responds by writing to the Read_Return register of the remote device with the contents of the register specified by the address. The Xilinx firmware handles this register automatically, as it supports SBus reads of remote registers mapped to SBus addresses 0x0001.00C0 through 0x0001.00FF.

Bit	Description
D0–7	Address of the SCD-FOI register to be read by the remote device.

Table 24. The FOI_Read_Trigger Register

FOI_Flow_Control Register

The FOI_Flow_Control register is an 8-bit register at address 0x0C. When the remote device writes to this register with a one, it inhibits burst writes of SBus DMA data to the remote device, thus providing flow control. To enable burst writes, the remote device writes a zero to this register.

Bit	Description
D0	A value of 1 inhibits burst writes. A value of 0 enables them.
D1–7	not used

Table 25. The FOI_Flow_Control Register

FOI_Burst Register

The FOI_Burst register is an 8-bit register at address 0x0D. Data written to this register is passed to the host computer main memory using DMA, provided that the SBus registers are configured for DMA. Reading this register from the remote device returns the last data written to the SBus Burst_Debug register, as an aid to hardware debugging.

Bit	Description
D0–7	Burst-write data from remote device.

Table 26. The FOI_Burst Register

FOI_Flag Register

The FOI_Flag register is an 8-bit register at address 0x0F. Data written to this register can be read from the SBus by means of the remote flag register. When the remote device writes to this register, the FLAGVALID bit of the Interrupt_Source register is set, allowing a host interrupt to be generated.

Bit	Description
D0–7	Flag data from remote device.

Table 27. The FOI_Flag Register

Specifications

The SBus Configurable DMA Fiber Optic Interface conforms to the following specifications.

SBus Compliance

Number of slots:	1
Transfer size	1, 2, 4, 8, or 16 words per transfer, up to the largest size accommodated by the host computer
DVMA master	
SBus memory space	approx. 66K bytes
Clock rate	25 MHz

Device Data Transfer

Protocol	Synchronous stream
Buffers	1 KB FIFO for input, 1 KB FIFO for output

Software

Drivers for Sun OS Version 4.1.x and 5.x (Solaris 1.x and 2.x)

Power

5 V at 2 A

Environmental

Temperature	Operating: 10 to 40 C Nonoperating: -20° to 60° C
Humidity	Operating: 20 to 80% noncondensing at 40° C Nonoperating: 95% noncondensing at 40° C

Physical

Dimensions	3.4" x 5.68" x 0.5"
Weight	4.9 oz.

Appendix A *ioctl()* Parameters

Engineering Design Team recommends that application programs use the software library interface described in the section entitled "Input and Output" beginning on page 5. However, if necessary, the following *ioctl* parameters may be useful for application programs. Others may be defined, but are used for Engineering Design Team's internal purposes only.

EDTx_RTOUT

Set (**S**) or get (**G**) the timeout lengths for reads from the SCD. The timeout units are hundredths of seconds. Provide an unsigned short as an argument.

EDTx_WTIMEOUT

Set (**S**) or get (**G**) the timeout lengths for writes to the SCD. The timeout units are hundredths of seconds. Provide an unsigned short as an argument.

SCDS_STATx_SIG

Set the signal to send to the current process upon receipt of a *STAT* interrupt. Replace *x* with an integer 1–4, specifying one of *STAT* bits 1–4. This *ioctl()* parameter enables interrupts so the SCD can receive the attention signal, even when I/O is not in progress. Set the signal to zero to disable the signal upon interrupt. Provide an unsigned integer as an argument.

EDTx_DEBUG_LEVEL

Set (**S**) or get (**G**) the debug level. A value of 1 reports driver status when *modinfo* (Solaris) or *modstat* (SunOS 4.1.x) is run. Other values are used for purposes internal to EDT. Provide an unsigned short as an argument.

EDTx_DEBUG_INTR

Set (**S**) or get (**G**) the debug interrupt level—1 generates full driver tracing and may interfere with normal driver operation. Provide an unsigned short as an argument.

SCDx_OUT_EMPTY, SCDx_OUT_FULL, SCDx_IN_EMPTY, SCDx_IN_FULL

Access FIFO control registers. This should never be done without specific direction from EDT, these registers control the internal operation of the SCD and are set to the proper values for your computer by the driver. These parameters require the address pointed to (the third parameter to *ioctl*) to contain unsigned integers (32 bits). Provide an unsigned short as an argument. Provide an unsigned integer as an argument.

SCDx_COMMAND

Set or get the value of the command register. Typically used to set the data polarity and the transmit clock source. The DMA direction and enable is accomplished by the read and write system calls. Provide an unsigned character as an argument.

SCDG_DATA_PATH_STATUS

Get the value of the data path status register. This registers reflects the state of the SCD FIFOs. Provide an unsigned character as an argument.

SCDx_DIR_WHEN_REGISTER

Set or get the value of the named register at the start of or the end of subsequent *read* or *write* system calls. Substitute **S** or **G** for *x*. Substitute **READ** or **WRITE** for *DIR*. Substitute **START** or **END** for *WHEN*. Substitute the register name for *REGISTER*. This word is valid only until the next *open()* of the SCD. Provide an unsigned character as an argument.

SCDS_DEF_SHORT_XFER

Set short transfer timeout mode. Provide an unsigned short as an argument.

Set **SCDS_DEF_SHORT_XFER** to a nonzero number to eliminate timeout errors. Upon timeout in this mode, *read* or *write* system calls return the size of the transfer instead of -1, and also put the transfer size into the `aio_result` structure for asynchronous I/O.

Set **SCDS_DEF_SHORT_XFER** to a number greater than one to reset the timeout counter at the start of each transfer.

If **SCDS_DEF_SHORT_XFER** is set to one, and either **EDTS_RTIMEOUT** or **EDTS_WTIMEOUT** are set, timeouts start during read or write operations only when input or output is not already in progress. The timeout is canceled if a transfer completes and no other asynchronous requests are queued.

EDTS_TERMINATE

Shut down ring buffers and terminate DMA on all outstanding I/O requests. No argument is required.

SCDx_FOI_CONFIG Set or get the value of the configuration register. Provide an unsigned character as an argument.

SCDx_FOI_INTERRUPT_STATUS

Set or get the value of the interrupt status register. Provide an unsigned character as an argument.

SCDx_FOI_INTERRUPT_ENABLE

Set or get the value of the interrupt enable register. Provide an unsigned character as an argument.

SCDG_FOI_INTERRUPT_SOURCE

Get the value of the interrupt source register. Provide an unsigned character as an argument.

SCDG_FOI_LOCAL_RECEIVER_ERROR

Get the value of the local receiver error register on the local device. Provide an unsigned character as an argument.

SCDG_FOI_REMOTE_ACK_ERROR

Get the value of the remote ack error register on the local device. Provide an unsigned character as an argument.

SCDG_FOI_REMOTE_READ

Get the value of the remote read register on the local device. Provide an unsigned character as an argument.

SCDG_FOI_REMOTE_FLAG

Get the value of the remote flag register on the local device. Provide an unsigned character as an argument.

EDTG_CUR_ADDR

Reads the contents of the current DMA address register. Provide an unsigned integer as an argument.

EDTG_CUR_COUNT

Reads the contents of the current count register. Provide an unsigned integer as an argument.

EDTG_NEXT_ADDR

Reads the contents of the next DMA address register. Provide an unsigned integer as an argument.

EDTG_NEXT_COUNT

Reads the contents of the next count register. Provide an unsigned integer as an argument.

